Debugging Tutorial

The purpose of this tutorial is to get you started debugging C programs using the **DDD** debugger.

Step 1: Compiling for debugging

The program you wish to debug must be compiled with the `-g` option. The debugger needs extra information (symbol table) from the compiler to associate machine instructions with source code lines.

We will use the sample program provided in the DDD documentation as a running example. So, make a copy of sample.c and compile it as follows:

```
gcc -g -o sample sample.c
```

Step 2: Starting DDD

You can start DDD by simply typing `ddd` at the command line. Then to load the executable select **File->Open Program** and navigate to the executable file you want to debug (`sample` in our example).

Alternatively, you can start DDD by typing `ddd sample` (assuming `sample` is in the current directory).

At this point you should see the source code for `sample.c`

Step 3: Setting a breakpoint

One of the most common reasons to use a debugger is to step through the code one line at at time to see exactly how variables are changing. The next thing we need to do is to tell DDD where we want to stop when running the program. This is called a **breakpoint**

There are several ways to set a breakpoint.

1. Left click on the line of code you want to stop on, then click on the stop sign button labeled Break in the tool bar. A stop sign should appear on the line of code you positioned the cursor on.
2. Position the mouse pointer over the line of code you want to stop on and hold the right mouse button down. You should see a menu in which you can select the "Set breakpoint" entry. Again, a stop sign should appear on the appropriate line of code.
3. Select the name of a function and click on the stop sign in the tool bar. A stop sign will appear beside the first executable line in that function.

Step 4: Running the program

This one is easy: click the Run button on the detached tool bar. The program will be run until the first breakpoint is reached. An arrow will appear beside the line that is to be executed next.

You can step through the program using the buttons on the detached tool bar or in the **Program** menu.

- **Next** Execute the next line of code stepping **over** functions.
- **Step** Execute the next line of code. If a function call, step **into** the function.
- **Cont** Continue executing the program until the next breakpoint is hit or the program terminates.

Step 5: Viewing the values of variables

- Roll the mouse over a variable and a box appears with the value of the variable in it.
- You can **display** the value of a variable by holding the right mouse button down and selecting the **Display** line. A new region of the main window appears with a box contain the name and value of the variable you selected. You can drag the boxes around in this window.
- If the variable you want to view is a pointer, you can display the dereferenced variable, by selecting **Display ***.
- You can view an element of an array, say `a[i]` by dragging the mouse across the whole variable and clicking on the display button.
- You can also display something by typing the variable into the box on the tool bar and clicking on the display button.

The Test

Find the bugs in the sample program by using the debugger. First you need to figure out what the program is supposed to do, and run the program to see what it is getting wrong. Hint: the DDD Reference in the Help menu walks you through finding the bug.

P.S.

This tutorial was written rather quickly. If you find errors, or if you think I have missed something important, please let me know.

Last modified: Thu Sep 13 19:32:55 EDT 2001

JStamp Tutorials - Install Java Tools

Please notice that the aJile tools must be installed *before* you can create the J2ME and aJile library references within JBuilder. If you already have JBuilder installed, that's OK, just install the J2ME and aJile files before you try to create the JBuilder library references.

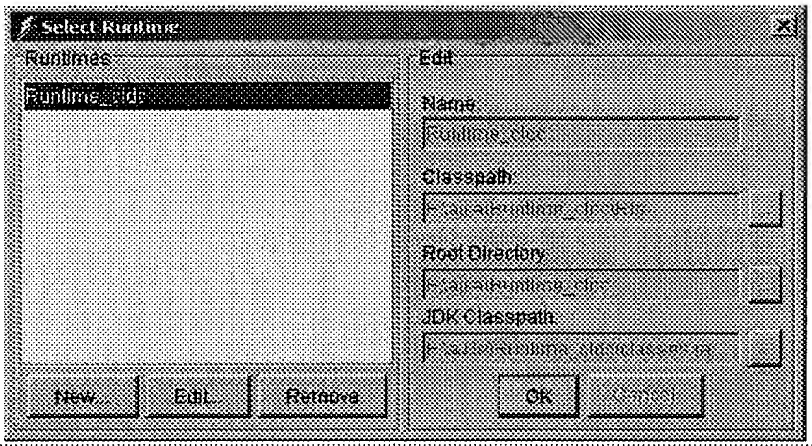
Install the Sun J2ME CLDC files & configure JemBuilder

Extract j2me_cldc-1_0_2-fcs-winunix.zip into a convenient place. I used e:\ (Be sure to check "use folder names" if you use Winzip!) This sets up my J2ME files in the root path e:\j2me_cldc\
The aJile J2ME runtime includes all the Sun J2ME base classes, so you don't need to add the Sun J2ME CLDC classes to your classpath. But you do want the Sun J2ME CLDC files for reference and for the CLDC base class javadocs. The aJile javadocs only include the aJile-specific classes.

The most recent version of J2ME locates the class files in a different folder from earlier releases. Probably just to keep us from getting too complacent and assuming everything would work as we expected. The current root classpath for j2me_cldc is E:\j2me_cldc\bin\common\api\classes

Start JemBuilder and make the toolbar selection Project->Properties. In the Embedded Runtime box, select Runtime_cldc. Make sure that the JDK classpath refers to the aJile runtime classes JAR file. At this time (May 2002) aJile supports the 1.0 version of J2ME CLDC, but the current version on the Sun website is 1.03. Don't use the Sun runtime in the aJile classpath until aJile makes the move to 1.03.

The JDK classpath tells JemBuilder where to find all the J2ME base classes. Check that the aJile paths are correct (they should be if you just installed the aJile tools). Here's what mine are:



Runtime_cldc

Name: Runtime_cldc
Classpath: [illegible]
Root Directory: [illegible]
JDK Classpath: [illegible]

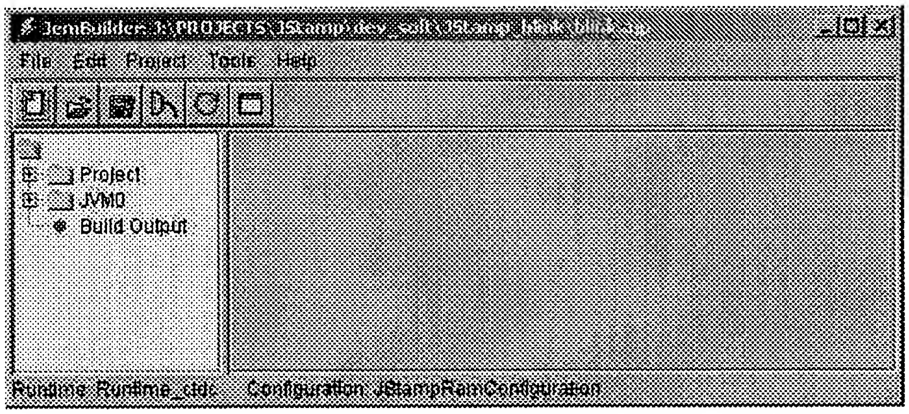
Install the aJile tools - Charade and JemBuilder

JemBuilder is the static linker and configuration tool. It converts a standard .class file into a .bin file suitable for execution on the aJile native Java controllers. JemBuilder is written in Java.

Charade is the JTAG loader/debugger. It communicates with your PC parallel port, through JTAG adapter hardware, to the aJile controller. Charade is written in C and has some native code for parallel port access. A Java version of Charade is being considered.

Charade is the reason that the aJile tools require a Win32 operating system.

Install the aJile tools provided on your JStamp CDROM. In my case I installed to e:\ajile. See if you can start JemBuilder:



We'll wait to start Charade until we can actually give it something to do.

Install JBuilder & set up a shortcut to the command line invocation

I installed to e:\JBuilderX where X is the version digit such as JBuilder6.

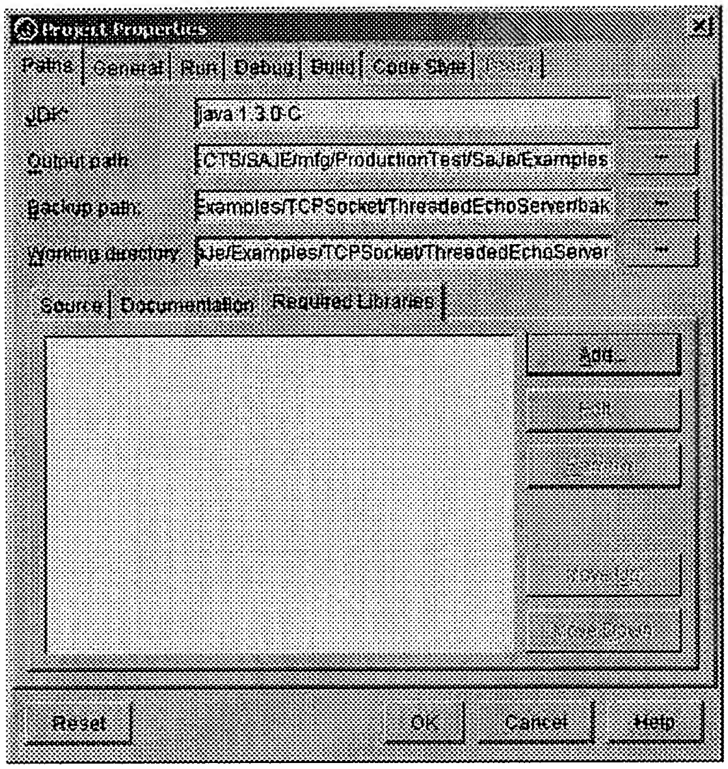
Start JBuilder and enter your license keys. Close JBuilder once you are convinced that it is starting up OK. We'll come back and do more with it in a minute. For the moment all we are going to use are the JDK files in the JBuilder "jdk1.3" subfolder.

Note: the current release of the aJile tools requires that you use a JDK1.3 -- aJile is not yet compatible with JDK 1.4 and you will get JemBuilder errors if you try to use 1.4.

Set up a shortcut to the command line invocation of JBuilder. In my case this is: E:\JBuilder6\bin\JBuilder.exe, instead of the Windows Start menu file E:\JBuilder6\binJBuilderW.exe
Starting JBuilder with JBuilder.exe leaves a command line window open, and lets you see runtime and diagnostic messages. This can be very helpful in diagnosing JBuilder plug-in failures, classpath problems, or other unexpected behaviors.

Configure JBuilder Project Properties

JBuilder needs to know where to find the aJile libraries. *The aJile runtime includes implementation of J2ME CLDC base classes so you don't need to also include the Sun J2ME CLDC as a library, just include the aJile runtime.* Create new required libraries by clicking the JBuilder Project menu, select properties, then the required libraries tab. In a new JB install, there will not be any libraries in the list:


Project Properties
Paths | General | Run | Debug | Build | Code Style
Source | Documentation | Required Libraries
Add...
Edit...
Remove
Move Up
Move Down
Reset
OK
Cancel
Help

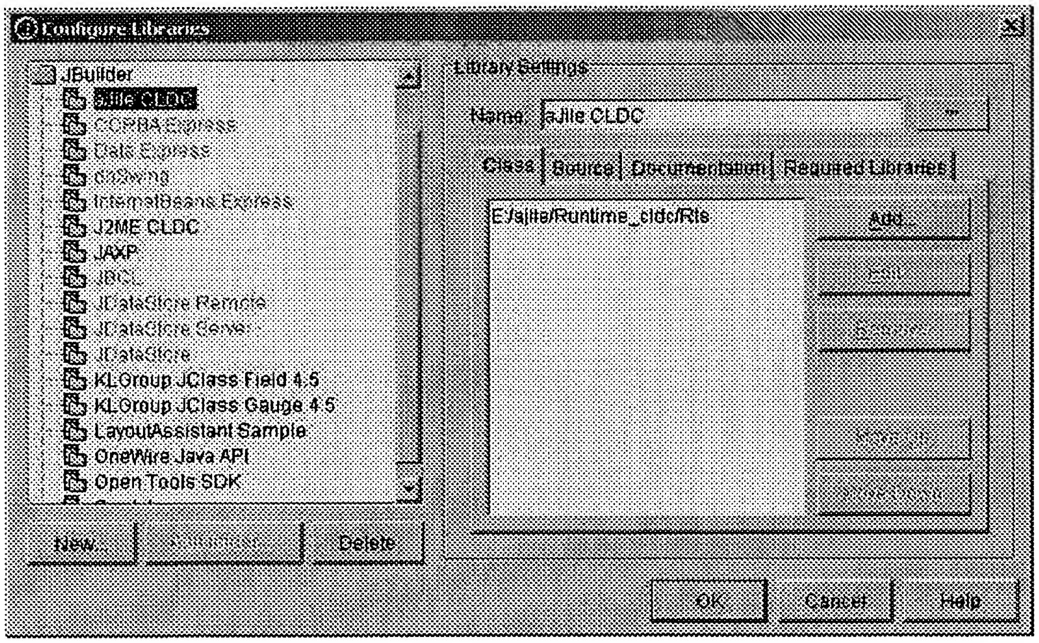
JDK: java 1.3.0-C

Output path: CTS/SAJE/mfg/ProductionTest/SaJe/Examples

Backup path: Examples/TCPSocket/ThreadedEchoServer/bak

Working directory: aJe/Examples/TCPSocket/ThreadedEchoServer

Click on Add, then New, and you can enter the name and then browse to the parent folder of the library classes. I created an aJile CLDC library:



Name: aJile CLDC

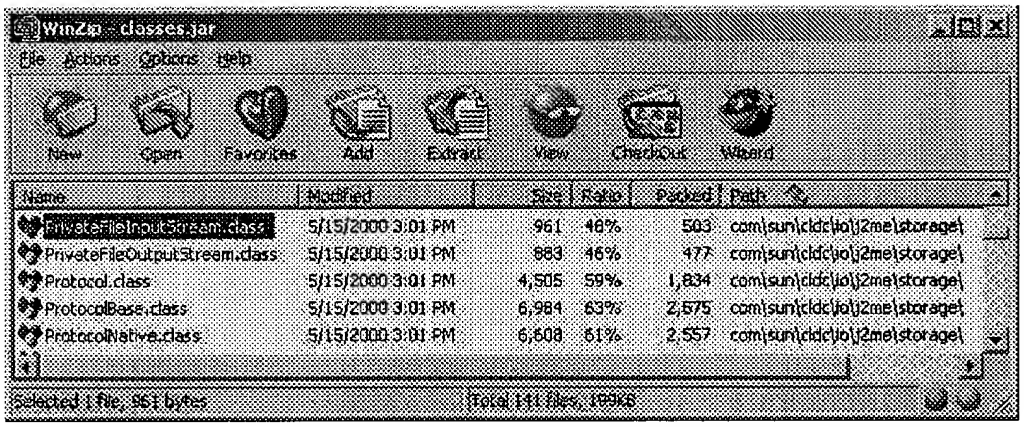
E:/ajile/Runtime_cldc/Rts

What is the aJile runtime_cldc/classes.jar file?

You may notice that aJile also provides a "classes.jar" file in the runtime_cldc folder. Don't use this, since it has been deprecated. It is only there since it includes some storage classes which Sun removed from the current J2ME CLDC release. J2ME CDC will add these back in, plus a more J2SE-like file system.

Notice that I'm using WinZip to open the JAR file - this is because a JAR file (JAR is an acronym for Java ARchive) is in WinZip format, and WinZip is a perfectly acceptable tool for working with JARs.

For the time being, classes.jar includes what flash file system suppport there is. Here are the storage classes:


WinZip - classes.jar
File Actions Options Help
New Open Favorites Add Extract View CheckOut Wizard

Name	Modified	Size	Ratio	Packed	Path
PrivateFileInputStream.class	5/15/2000 3:01 PM	961	48%	503	com\sun\cldc\io\j2me\storage\
PrivateFileOutputStream.class	5/15/2000 3:01 PM	883	46%	477	com\sun\cldc\io\j2me\storage\
Protocol.class	5/15/2000 3:01 PM	4,505	59%	1,834	com\sun\cldc\io\j2me\storage\
ProtocolBase.class	5/15/2000 3:01 PM	6,984	63%	2,575	com\sun\cldc\io\j2me\storage\
ProtocolNative.class	5/15/2000 3:01 PM	6,608	61%	2,557	com\sun\cldc\io\j2me\storage\

Selected 1 file, 961 bytes   Total 111 files, 109KB

If you must use these, you can create another required library, referencing this classes.jar file, *but be sure it is the last entry in your required libraries* since it has class names which are identical to those in the RTS folder, and you want the ones in the RTS folder to be found first.

JB remembers these definitions and they are easily included in subsequent projects. You can also add these to the "default project" description (also accessed from the toolbar Project menu).

Update any Windows System parameters (a very helpful, often overlooked tip!)

NT4's System properties (Control Panel->System->Environment) lets you set parameters such as a global classpath and system path.

I add e:\JBuilder6\jdk1.3\bin; to the beginning of the PATH environment variable, so that from a command prompt, the JDK1.3 java, javac, javadoc, etc are the *first* "java" executables found. This avoids the common mistake of invoking a browser or Windows system "java" executable from the command line. I also set my CLASSPATH here to include e:\JBuilder6\jdk1.3\lib\comm.jar;

Win2000 has a similar feature in Settings->Control Panel->System->Advanced->Environment Variables.

Check the version of java which executes from a command prompt

Open a command prompt and type "java -version", and if your system path is set properly, you should see version 1.3, like this (JB4 is shown, other versions are similar):

```
E:\JBuilder4\commapi\samples\BlackBox>java -version
java version "1.3.0"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.0-C)
Java HotSpot(TM) Client VM (build 1.3.0-C, mixed mode)
```

If you get some other Java version, then there is something else in your environment which is referencing another "java" installed on your PC. This could be part of an internet browser, an old Java tool installation, or something else. Look at your system PATH value and change it so that the new JBuilder jdk 1.3 bin folder is in the path before any other java references.

Install the Sun Java COMM API

Extract javacomm20-win32.zip into a convenient place. I used e:\JBuilder6\commapi. (Be sure to check "use folder names" if you use Winzip!) Now in that folder (e:\JBuilder6\commapi for me) open the file PlatformSpecific.html, it has instructions for installing the commapi files. Here's the short version, edited for my case.

Copy e:\JBuilder6\commapi\win32com.dll to e:\JBuilder6\jdk1.3\bin

Copy e:\JBuilder6\commapi\comm.jar to e:\JBuilder6\jdk1.3\lib

(Apparently JBuilder looks in the folder e:\JBuilder6\jdk1.3\jre\lib\ext -- if you copy comm.jar there, then you don't need to add it to your classpath -- JBuilder and the command line java tools will find it. However some programs aren't entirely happy with this and will generate errors, so add it to your classpath explicitly and leave it in the jdk1.3\lib folder only)

Copy e:\JBuilder6\commapi\javax.comm.properties to e:\JBuilder6\jdk1.3\lib directory. Failure to do this will result in the inability to enumerate your PC serial ports - instead of COM1, COM2 etc, you will see 'serial0' which are just the name placeholders, not real serial ports. You can't access 'serial0'.

Run BlackBox to test the COMMAPI.

Run BlackBox to test the COMMAPI

Add the comm.jar file to your classpath, in my case this can be done with a
```
set CLASSPATH=e:\JBuilder6\jdk1.3\lib\comm.jar;%CLASSPATH%
```

Then type "java BlackBox" at a command prompt and BlackBox should claim your PC's serial port(s) and start executing. If not, it is typically because the comm.jar file is not in your classpath. If your serial ports can't be enumerated, then javax.comm.properties is not in the correct place.

Troubleshooting

Recently I was startled to see this error message on my old reliable NT4 system:

```
E:\JBuilder6\commapi\samples\BLACKBOX>java BlackBox
Exception in thread "main" java.lang.NoClassDefFoundError: javax/comm/CommPort

E:\JBuilder6\commapi\samples\BLACKBOX>
```

What could I have done? Things used to work fine. I hadn't changed anything (or so I thought). The problem turned out to be that I had "cleaned up" some duplicate files on my system. One of them was comm.jar, and so my classpath no longer could find it. Simple but sometimes maddening. Easily fixed.

If Blackbox won't run, it is typically because the comm.jar file is not in your classpath. If your serial ports can't be enumerated, then javax.comm.properties is not in the correct place.

Get to a command prompt and type "set" followed by an enter key. You should see a list of all the environment variables. Verify that CLASSPATH and PATH are correct. If not, set them here.

Previous: Get needed files	Tutorial Home	

aJile Systems: Low-Power Direct-Execution Java^{TM} Microprocessors for Real-Time and Networked Embedded Applications

David S. Hardin
aJile Systems, Inc.
http://www.ajile.com

Abstract

The Java environment, with its platform neutrality, simplified object model, strong notions of safety and security, as well as multithreading support, provides many advantages for a new generation of networked embedded and real-time systems. However, the large size, nondeterministic behavior, and poor performance of the first generation of embedded Java implementations have hampered the acceptance of Java in the real-time and embedded worlds. aJile Systems has developed a low-power hardware implementation of the Java Virtual Machine which makes real-time embedded Java a practical reality. aJile's hardware provides direct support for the entire JVM instruction set and thread model, obviating the need for a Java interpreter or Just-In-Time (JIT) compiler, as well as the traditional Real-Time Operating System (RTOS). aJile's hardware technology also supports multiple JVM contexts executing on the same CPU, enhancing safety and security by guaranteeing space and time allotments for multiple Java applications. Combined with a Java 2 Micro Edition (J2ME) runtime and a back-end target build tool, these technologies constitute an efficient platform for the development of real-time embedded applications entirely in Java.

1. Introduction

The embedded and real-time system marketplace is exploding in the "post-PC" era, especially as more and more devices are becoming Internet-enabled. Networked, real-time embedded systems are becoming common in markets such as telecommunications, industrial automation, home and building control, automotive systems, and medical instrumentation. The software content of these networked devices is soaring, putting a significant strain on scarce development resources. The real-time and embedded developer also faces an extremely heterogeneous processing environment, with a plethora of processors, operating systems, and peripheral device types. Thus, engineers are increasingly looking to Java technologies to provide a more productive, portable development environment for real-time and embedded systems. These technologies include the Java object-oriented programming language; the Java Virtual Machine; as well as a large selection of runtime class libraries.

2. The Java Platform

The Java platform debuted in 1995 as a desktop and server language environment, although it was originally developed as an object-oriented programming language for embedded devices. The platform neutrality, simplified object model, strong notions of safety and security, as well as multithreading support, of the Java platform provides many advantages for a new generation of networked embedded and real-time systems.

2.1 The Java programming language

The Java object-oriented programming language [1], with its familiar C-like syntax and simplified object model, is easier to master than C++, and has been shown to be 25% - 40% more productive. Java's lack of user-manipulable pointers and automatic memory management are frequently cited as key factors in improved software productivity and robustness. The Java language also supports priority-preemptive multithreading, and the `synchronized` keyword provides a particularly elegant means of enforcing mutual exclusion synchronization.

2.2 The Java Virtual Machine

Java execution is supported by a standardized Java Virtual Machine (JVM) to which Java programs are compiled [7]. This design enables Java to be highly portable since Java programs can run on any system that supports the JVM. Moreover, the underlying target computer system implementation is insulated from the application by the JVM, thus enhancing the safety and security of mobile code. Indeed, the entire Java environment is designed to support network-distributed computing.

The JVM "bytecode" instruction set is a stack-based 32-bit architecture, with a number of unique instructions, such as virtual method invocation with lock detection. In a typical Java runtime environment, portable applications or applets compiled into Java Virtual Machine bytecodes are interpreted by a software JVM implementation, Just-In-Time (JIT) compiled to native machine code, or directly executed by a Java microprocessor.

A Java microprocessor is the most space- and time-efficient vehicle for executing Java bytecode. Interpreters are slow (up to an order of magnitude slowdown) and require memory to store the interpreter code. JIT's require sizable memory areas to hold the translated code, and execution in a JIT environment suffers from cache miss effects if control transfers to a non-translated code block. However, building such a processor is a nontrivial task, as the JVM instruction set is quite complex, much more so than even traditional Complex Instruction Set Computers (CISC's) such as the Intel x86 and Motorola 680x0 families.

2.3 Java for mobile devices

The J2ME Connected, Limited Device Configuration (CLDC) [5] specifies a Java 2 subset for memory-constrained (less than 512 KB available for the Java environment) devices with lower-speed and potentially intermittent network connectivity, such as mobile phones and Personal Digital Assistants. The Connected Device Configuration (CDC) is targeted at somewhat less-constrained devices, such as set-top boxes and other "plug-in-the-wall" network appliances. In addition to these standard configurations, a number of vertical market profiles can be defined for J2ME devices. One particularly important J2ME profile is the Mobile Information Device Profile (MIDP) [6], which, amongst other features, defines a minimal set of graphical API's for information appliances such as mobile phones, Personal Digital Assistants, etc. The software architecture for a typical J2ME application is shown in Figure 1.

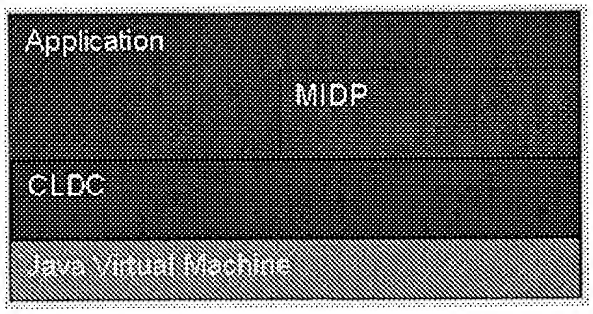


Figure 1. J2ME software architecture.

2.4 The Real-Time Specification for Java

The dynamic nature of the Java runtime environment is one of its greatest strengths in the traditional desktop and server world. However, the nondeterminism introduced by garbage collection, runtime class resolution, etc., is a real problem for real-time developers. Thus, the Real-Time for Java Expert Group (of which aJile is a member) was formed in March 1999 under the Java Community Process to create a Real-Time Specification for Java (RTSJ) [2]. The RTSJ provides enhancements to the Java Language Specification [4] and Java Virtual Machine Specification [7] in seven key areas: Thread Scheduling and Dispatching; Memory Management; Synchronization and Resource Sharing; Asynchronous Event Handling; Asynchronous Transfer of Control; Thread Termination; and Physical Memory Access. As an example of these enhancements, the RTSJ defines new types of memory areas separate from the Java heap, including `ImmortalMemory` and `ScopedMemory`. Real-time threads can create Java objects in these alternative memory areas in the normal fashion (i.e., through the `new` operator), but since these objects do not reside in the Java heap, object access does not suffer from nondeterministic garbage collection pauses.

3. The aJile architecture for efficient real-time embedded Java systems

aJile Systems was formed to meet the need for low-power real-time embedded object-oriented application deployment. The aJile Java architecture was designed with an embedded systems focus, and thus maximizes the amount of runtime data that can be ROM'ed, and eliminates runtime instruction stream modification ("quickizing"). aJile CPU's directly supports the Java thread model in hardware, yielding extremely fast thread switching times. The aJile architecture also defines a set of "extended" instructions for physical hardware interfacing and other systems programming tasks; these extended instructions are not available to untrusted dynamically downloaded code.

3.1 The JEM2 direct execution Java microprocessor core

The aJile Systems JEM2 is a second-generation low-power (1 mW/MHz) direct execution Java microprocessor core. The compact, low-power design of the JEM2 makes it particularly well-suited as a microcontroller core in application areas such as telecommunications, automotive, and industrial automation. JEM2 supports 32, 16, or 8 bit external data buses, and provides a standard IEEE 1149.1

(JTAG) test interface. With JEM2, real-time embedded developers can use the popular object-oriented Java language, with its proven productivity advantages, to produce applications that are as space- and time-efficient as those written in languages such as C for other microcontroller platforms. Benchmarking results indicate that performance of Java on the JEM2 is comparable to the same application written in C on a comparable low-end 32-bit microcontroller; and 4-5 times the performance of embedded Java on other "standard" 32-bit embedded processors (all results normalized to the same clock frequency). Additionally, the JEM2 is up to 20 times more power-efficient per unit of Java performance than other embedded Java implementations.

The JEM2 architecture is shown in Figure 2.

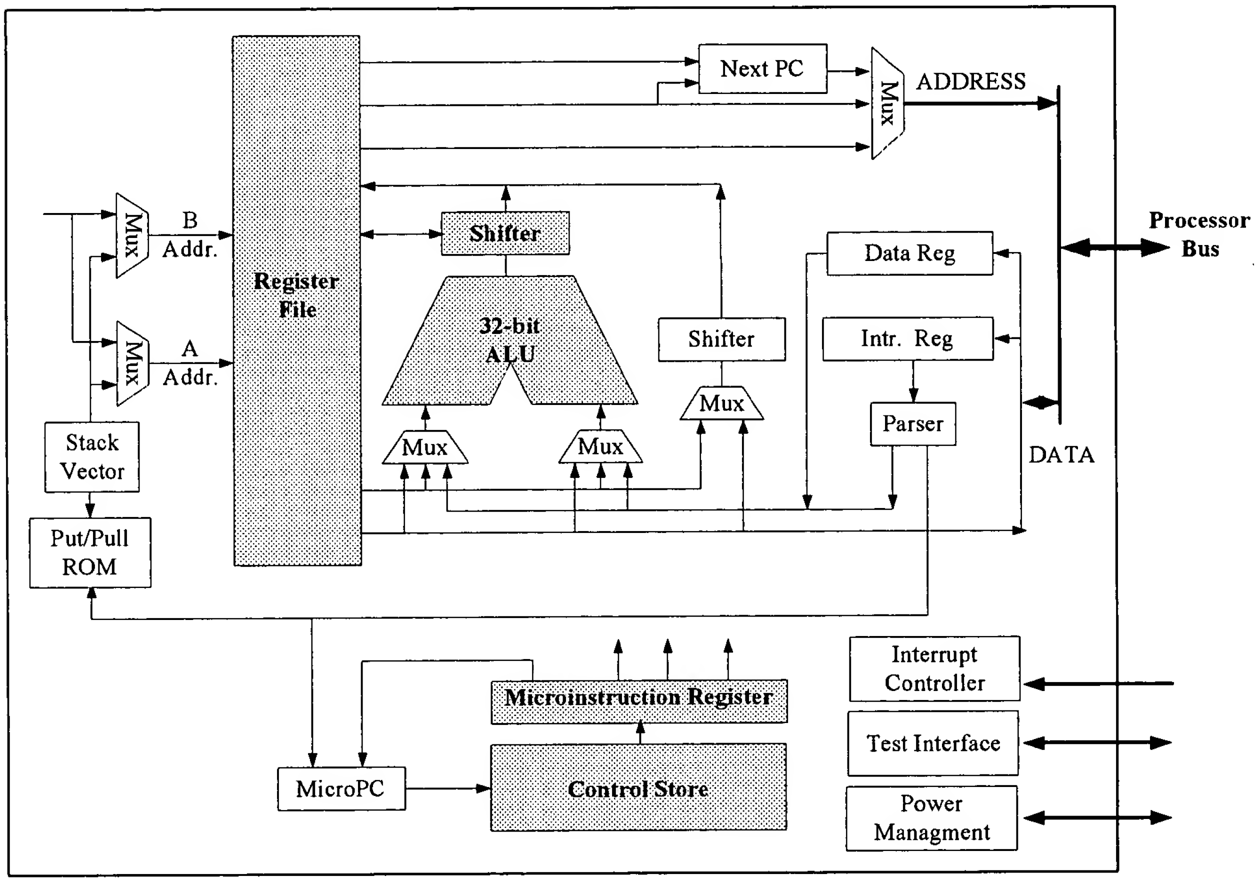


Figure 2. JEM2 core architecture.

The JEM2 core implements the entire JVM instruction set in silicon; the only two bytecodes that trap immediately to software are `multianewarray` and `athrow`. Obviously, operations like class loading are handled in software, but once resolution has occurred, execution of bytecodes like `invokevirtual` are done as single JEM instructions, including lock detection.

3.2 Hardware support for real-time Java threads

One of the unique features of the aJile architecture is its hardware support for real-time Java threads. Concurrency control is deeply ingrained in the Java Virtual Machine specification; elementary operations such as the method invocation instructions require the acquisition of a lock if the target method is declared as `synchronized`. Thus, aJile CPU's implement the basic synchronization and thread scheduling routines in microcode. This means, for instance, that the `yield()` primitive in `java.lang.Thread` is a single extended bytecode. Several benefits accrue from this approach. First, aJile CPU's require no Real-Time Operating System (RTOS) kernel, thus saving kilobytes of memory. Further, multithreading on aJile CPU's is extremely fast. For example, the time required to execute a `yield()` and resume a different thread is approximately 500 nanoseconds on a 100 MHz aJile CPU. Additionally, aJile hardware supports periodic thread dispatching, and also implements priority inversion control.

3.3 A Java runtime written entirely in Java

A unique feature of the aJile runtime environment is that it is programmed completely in the Java language, including substitutes for the many "native methods" in the Java runtime system. This is made possible by the JEMBuilder application build tool, which provides the substitution of certain method invocations with aJile extended bytecodes.

```
void SerialInterruptHandler() {
  while  (true) {
    byte isrVal = rawJEM.getByte(XR16C850.ISR);

    //--- have all pending interrupts been processed? ---
    if ((isrVal & 0x01) == 1) return;

    //--- decode which interrupt is pending ---
    isrVal = (byte) ((isrVal >> 1) & 0x7);
    switch(isrVal) {
        << cases elided >>
        case 3: // LSR
        byte lsrVal = rawJEM.getByte(XR16C850.LSR);
        boolean OE  = (lsrVal & 0x02) == 0x02;
        boolean PE  = (lsrVal & 0x04) == 0x04;
        << etc. >>
        if ((lsrVal & 0x02) == 0x02)
{serialThread.pendSerialEvent(SerialPortEvent.OE,OE,m_OE);}
        if ((lsrVal & 0x04) == 0x04)
{serialThread.pendSerialEvent(SerialPortEvent.PE,PE,m_PE); }
        << etc. >>
        m_OE = OE; m_PE = PE;
        break;
      default:
        rawJEM.set(0x01200000,isrVal);  // drive the external interrupt pin
    } // end switch
  } // end while
} // end serial interrupt handler
```

Figure 3. Device driver code in Java.

With this approach, there is no need for an assembler, and developers can create Java implementations for the aJile extended bytecode classes (e.g. physical memory access) in a host simulation environment. The result is that the entire aJile Java runtime is written in Java, thus easing both testing and maintenance.

Figure 3 provides an example of device level programming in Java. In addition to exhibiting direct access to device registers (via the `rawJEM.*` methods), this example also shows how device-level code can notify waiting user threads (through the `serialThread.pendSerialEvent()` calls).

Note that there is no need to abandon object-oriented Java programming practice in order to do system-level programming for the aJile environment.

3.4 Memory management

The Java Virtual Machine specification includes several bytecodes for memory allocation. However, the Java environment has no `free()` primitive; rather, memory reclamation is automatic. Accordingly, the aJile environment implements the memory allocation bytecodes as instructions, and implements a simple mark-and-sweep garbage collector in software.

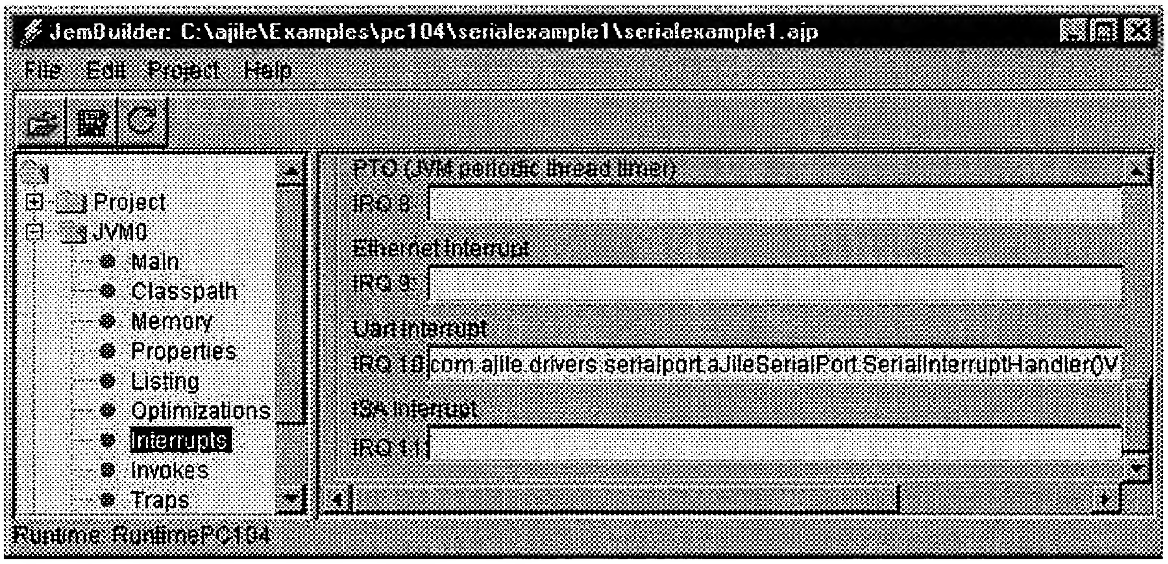
The aJile garbage collector is implemented as a Java thread, and utilizes Java synchronization to assure that the heap it is collecting does not get into an inconsistent state. Heaps in the aJile architecture can be allocated per-thread; thus, the garbage collector can be preempted very quickly (< 1 μsec on a 100 MHz aJ-100) in order to allow a real-time thread (which uses a separate, non-garbage collected heap) to execute.

3.5 Interrupt and trap handling

The virtual nature of the JVM specification means that low-level mechanisms such as interrupt and trap handling are left unspecified. So, the aJile architects have provided our own implementation. In keeping with the theme of writing all software in Java, interrupts and traps are handled by static Java methods, which execute on an "executive", or supervisor, stack. The executive mode of the processor can be thought of as a single highest-priority thread that has its own stack and heap, and whose methods are the various trap and interrupt handlers. As one would expect, maskable interrupt handlers can be preempted by the occurrence of other, higher priority interrupts, if interrupts are enabled in executive mode (by default, interrupts are disabled when the processor enters executive mode). When the higher priority handler is invoked, a new stack frame is pushed onto the executive stack. Exit from that handler returns control to the preempted handler. Executive mode execution completes when the outermost handler returns.

3.6 Tool support

The aJile development environment allows the use of any off-the-shelf compiler that produces Java standard class files. Thus, developers are free to use whatever Java Integrated Development Environment (IDE) they wish. The only custom elements in the aJile tool chain are the JEMBuilder graphical build tool, which produces JEM format binary images from input class files, and low-level debugging tools.



PTO (JVM periodic thread timer)
IRQ 8

Ethernet Interrupt
IRQ 9

Uart Interrupt
IRQ 10 com.ajile.drivers.serialport.aJileSerialPort.SerialInterruptHandler()V

ISA Interrupt
IRQ 11

Runtime: RuntimePC104

Figure 4. The JEMBuilder application builder.

Most Java implementations rely on runtime class resolution, but this is not necessary if all the necessary classes are present at build time. The aJile JEMBuilder application, itself written in Java, computes the class dependencies for a standard format Java class file, and can create target RAM and ROM memory images only for those classes that are needed. JEMBuilder can also eliminate unused methods, fields, and constants, resulting in significantly reduced memory requirements (often by a factor of 10) for dedicated Java applications. Java's safety and security orientation, particularly the avoidance of pointer manipulation, helps make this build-time analysis possible.

JEMBuilder also includes a number of additional capabilities, including the substitution of named static methods with extended bytecodes, as well as the mapping of named Java methods to interrupt and trap handlers (as shown in Figure 4). The former capability allows off the-shelf compilers to be used; only JEMBuilder need know about the extended bytecodes. Finally, JEMBuilder presents these capabilities in a user-friendly way using Java Swing graphical components.

The most innovative characteristic of the aJile tool chain is that it allows all application and system software to be written in the Java programming language (or other programming languages that compile to standard Java class files). This allows pure object-oriented design techniques to be brought all the way down to the "bare metal". This "low-level programming in a high-level language" philosophy eases integration, testing, and maintenance, as well as promoting reuse at the system software level.

Figure 4 provides a screenshot of JEMBuilder, highlighting the mapping of interrupts to Java handlers.

3.7 Instruction set customization

aJile CPU's generally provide a writeable control store, allowing the instruction set to be customized for particular applications. As with other extended bytecodes, these user-defined instructions are introduced into the instruction stream by the JEMBuilder tool via named static method substitution; no compiler changes are required.

3.8 Multiple concurrent Java Virtual Machines

The aJile architecture provides hardware support for the concurrent execution of multiple independent Java applications (i.e., multiple `main()`'s) in a deterministic, time-sliced schedule with full memory protection. Within its bounded execution interval and memory space, each application environment can employ its own threading and memory management policies without threat of intervention by other faulty or malicious applications. Additionally, each JVM has its own executive mode; this allows JVM's to be assigned their own interrupts, which pend while that JVM is suspended. This "Multiple JVM" (MJM) capability takes the Java "sandbox" security model to the next level, providing hard space and time boundaries for concurrent JVM execution.

4. aJ-100: A real-time embedded single-chip Java microcontroller

The aJile Systems aJ-100 [3] is a real-time embedded single-chip Java microcontroller based on the JEM2 core that integrates Java Virtual Machine (JVM) bytecode execution, real-time Java threading primitives, and Multiple Java Virtual Machine support, along with common embedded peripherals. An architectural overview of the aJ-100 chip is shown in Figure 5.

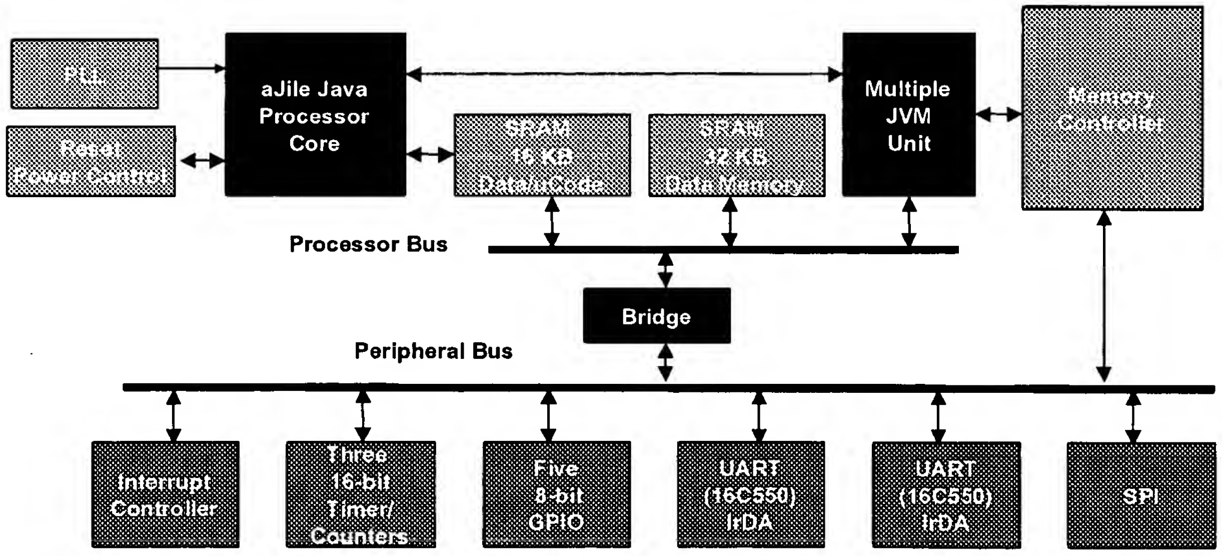


Figure 5. aJ-100 block diagram.

This integration allows developers to take advantage of the benefits of Java development in a compact low-power package with the processing performance to support embedded systems. With a low-power 32-bit Java core, on-chip memory and a set of peripherals, this real-time Java microcontroller platform is well-suited for smart mobile devices, consumer appliances, automotive applications, and networked industrial controllers.

By providing a "system on chip" approach, the aJ-100 allows developers to easily provide embedded Java functionality for their products. This is further aided by a bundled J2ME CLDC runtime, including all device drivers for the aJ-100 peripherals, as well as drivers for common external peripherals such as Ethernet, FLASH memory, and LCD display controllers.

The aJ-100 operates at clock rates up to 100 MHz over an industrial (-40C to +85C) temperature range. The aJ-100 is packaged in a 176-pin LQFP, as shown in Figure 6, and has been shipping since December 2000.

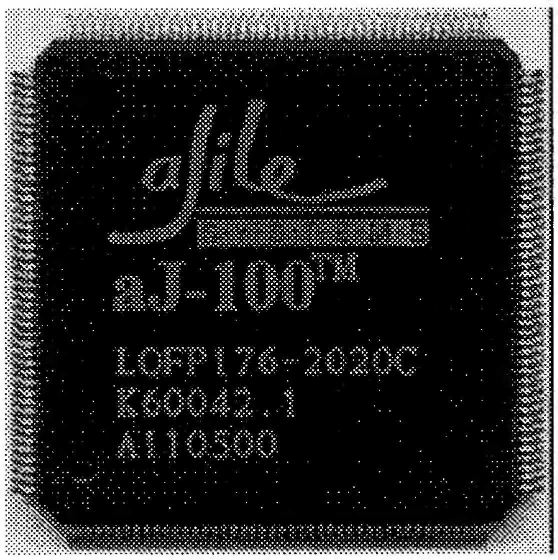


Figure 6. aJ-100 package (larger than actual size).

5. Conclusions

The aJile family of low-power single-chip embedded Java microcontrollers provides an efficient platform for embedded and real-time object-oriented software execution. The aJile CPU hardware provides direct support for the entire JVM instruction set and thread model, obviating the need for a Java interpreter or Just-In-Time (JIT) compiler, as well as the traditional Real-Time Operating System (RTOS). aJile's hardware technology also supports multiple JVM contexts executing on the same CPU, enhancing safety and security by guaranteeing space and time allotments for multiple Java applications. Combined with a Java 2 Micro Edition (J2ME) runtime and a back-end target build tool, these technologies allow the development of real-time embedded applications entirely in Java.

6. Acknowledgments

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. JEM is a trademark of Rockwell Collins, Inc. aJile and aJ-100 are trademarks of aJile Systems, Inc. All other trademarks are property of their respective holders.

7. References

[1] Ken Arnold, James Gosling, and David Holmes, *The Java Programming Language*, third edition, Addison-Wesley, 2000.

[2] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull, *The Real-Time Specification for Java*, Addison-Wesley, June 2000. Further information is available at http://www.rtj.org.

[3] David Hardin, *aJ-100: A Low-Power Java Processor*, Presentation at the Embedded Processor Forum, June 2000. Available at http://www.ajile.com/Documents/ajile-epf-2000.pdf.

[4] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification*, second edition, Addison-Wesley, 2000.

[5] JSR-30 Expert Group, Java 2 Platform, Micro Edition, *Connected, Limited Device Configuration*, Version 1.0, 19 May 2000. Available at http://java.sun.com/aboutJava/communityprocess/final/jsr030/index.html.

[6] JSR-37 Expert Group, Java 2 Platform, Micro Edition, *Mobile Information Device Profile*, Version 1.0, 1 September 2000. Available at http://java.sun.com/aboutJava/communityprocess/final/jsr037/index.html.

[7] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, second edition, Addison-Wesley, 1999.
Sun Microsystems, Inc., Java 2 Platform, Micro Edition Home Page, http://java.sun.com/j2me/.

[8] Sun Microsystems, Inc., The Java Community Process. Available at http://java.sun.com/aboutJava/communityprocess/java_community_process.html.